R-560-NASA/PR
December 1970

EXPERIENCE WITH THE EXTENDABLE COMPUTER SYSTEM SIMULATOR

D. W. Kosy

A Report prepared for

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
AND
UNITED STATES AIR FORCE PROJECT RAND

Rand
SANTA MONICA, CA. 90406

This research is sponsored by the National Aeronautics and Space Administration under Contract No. NAS-12-21-44 and the United States Air Force under Project Rand—Contract No. F44620-67-C-0045—monitored by the Directorate of Operational Requirements and Development Plans, Deputy Chief of Staff, Research and Development, Hq USAF. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of Rand, NASA or of the United States Air Force.

ERRATA

R-560-NASA/PR EXPERIENCE WITH THE EXTENDABLE COMPUTER SYSTEM SIMULATOR, D. W. Kosy, 12-70

Page 15 - Last paragraph entitled "AUTOMATIC SUMMARIES" - First line

Following "A computer system", add the word "simulator"

R-560-NASA/PR

December 1970

EXPERIENCE WITH THE EXTENDABLE COMPUTER SYSTEM SIMULATOR

D. W. Kosy

A Report prepared for

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
AND
UNITED STATES AIR FORCE PROJECT RAND

Rand
SANTA MONICA, CA. 90406

Rand maintains a number of special, subject bibliographies containing abstracts of Rand publications in fields of wide current interest. The following bibliographies are available upon request:

Africa • Arms Control • Civil Defense • Combinatorics
Communication Satellites • Communication Systems • Communist China
Computing Technology • Decisionmaking • East-West Trade
Education • Foreign Aid • Health-related Research • Latin America
Linguistics • Long-range Forecasting • Maintenance
Mathematical Modeling of Physiological Processes • Middle East
Policy Sciences • Pollution • Procurement and R&D Strategy
Program Budgeting • SIMSCRIPT and Its Applications • Southeast Asia
Systems Analysis • Television • Urban Problems • USSR
Water Resources • Weather Forecasting and Control

To obtain copies of these bibliographies, and to receive information on how to obtain copies of individual publications, write to: Communications Department, Rand, 1700 Main Street, Santa Monica, California 90406.

Published by The Rand Corporation

PREFACE

Although the value of performing simulation analyses for the design and evaluation of military and aerospace computer systems is recognized, and designers and users of such systems often desire to employ this tool, the difficulty and expense of developing sufficiently detailed models is often prohibitive. The Air Force, in particular, requires a powerful and flexible simulation capability to evaluate present and proposed systems. Accordingly, we have developed the Extendable Computer System Simulator (ECSS) to permit system models to be built more rapidly and at less cost than previously possible. Although the original development of ECSS was sponsored by NASA, work on it was continued under Air Force Project RAND. It has a particular relevance to work being done for the Air Force Logistics Command on the large hardware/software effort contained in the Advanced Logistics System (ALS).

ECSS is a special-purpose language for computer system modeling. The original design is presented in N. R. Nielsen, *ECSS: An Extendable Computer System Simulator*, The RAND Corporation, RM-6132-NASA, February 1970. The present Report discusses Rand experience with the initial version of the language, outlines its strengths and weaknesses, and offers some general principles for improving the language with respect to user capability and convenience.

This Report should be of interest to potential users of ECSS and to other designers of computer system modeling languages. The reader should be familiar with computer system concepts and terminology, and with discrete-event simulation. Some knowledge of SIMSCRIPT II programming language is required to completely understand the ECSS example program in Appendix B.

SUMMARY

A prototype version of the Extendable Computer System Simulator (ECSS) has been implemented to aid in constructing simulation models of computer systems. A specialized language is used to describe hardware, software, and system load. A service-routine package handles many of the housekeeping details of model control. The full power of SIMSCRIPT II is also available for extending ECSS capabilities. Advantages of ECSS over other languages include its natural, English-like input format, provisions for compact description of common computing system elements and operations, flexibility, extendability, modifiability, and provisions for economical simulation reruns.

Some weaknesses in the provided facilities have been noticed, however, in that certain features are either absent or less convenient than they should be. Summary reports of model structure, for example, are not produced, nor are statistics on model operation collected or reported. The user finds some mechanisms not as accessible or controllable as sometimes needed. Moreover, certain control-program models require a very awkward representation within ECSS.

Reflection on the strengths and weaknesses of this ECSS prototype stimulated a list of design factors for computer system simulators. Some considerations are: 1) declarative model specification is more convenient, but procedural specification more flexible; 2) powerful statements allow quick development of coarse models, but a number of simpler statements are required for more finely detailed models; 3) a service-routine executive must be easily accessible to allow the adaptability required of a computer system simulator; and 4) certain information is of interest in all computer system simulations and should be available on demand as preformatted reports from the simulator.

CONTENTS

FIGURES

I. INTRODUCTION

The Extendable Computer System Simulator (ECSS) is a prototype language, designed and implemented to investigate ways of making the simulation of complex computer systems a less formidable task. The need for a new language, free from the problems of using general-purpose languages and the drawbacks of existing computer system simulators was demonstrated in Neilsen, 1970 [1]. Our approach is to provide a convenient and natural means of describing computer system characteristics and computing processes while allowing the flexibility and power of a general-purpose simulation language.

Experience with several small models, written to test the initial version of the simulator, has indicated the soundness of the approach. In most cases, the models have been quickly and easily constructed. However, situations have also been found in which the language is not as useful as possible, revealing shortcomings in both the breadth and versatility of the facilities provided.

This Report describes the current capabilities of ECSS, discusses ECSS strengths and weaknesses, and prescribes some directions for further work. We first review the concepts of the language, and list the advantages of the ECSS approach. We then outline a number of cases where the current version of the language is not as helpful as it could be. Finally, we present several general principles, which have become clearer in retrospect, for the design of computer system modeling languages. Two appendices show the details of ECSS program processing, and provide a specific example of a complete ECSS simulation program.

II. REVIEW OF ECSS

Three elements of ECSS are fundamental in describing a computer system to be simulated: 1) the System Description section; 2) the Load Description section; and 3) the Service Routines.

SYSTEM DESCRIPTION

The System Description characterizes those system elements considered static in ECSS. Static elements are called *devices*, reflecting their usage as models of pieces of hardware. Declarations in this section specify the number of the various types of hardware, the names of these devices and names of groups of devices, the capacities and capabilities of the devices, the interconnection of devices, and the possible "software execution time overhead" incurred by simulated operation of some devices in performing certain operations.

To model different types of hardware, the user specifies ECSS devices to have the appropriate characteristics. All devices are viewed as collections of up to four *components* (see Fig. 1), and different device characteristics result from using different combinations of components. Each type of component represents a different device capability. For some hardware models the conjunction of several components may be meaningful, e.g., in defining a central processor to execute instructions, transmit data, and have core storage; but often, only one component is important, e.g., in defining a disk memory as only a quantity of storage space. The user may choose the combination of components necessary to model real equipment.

Device characteristics are further specified by setting parameters of the components. Each type of component has different parameters according to that component's

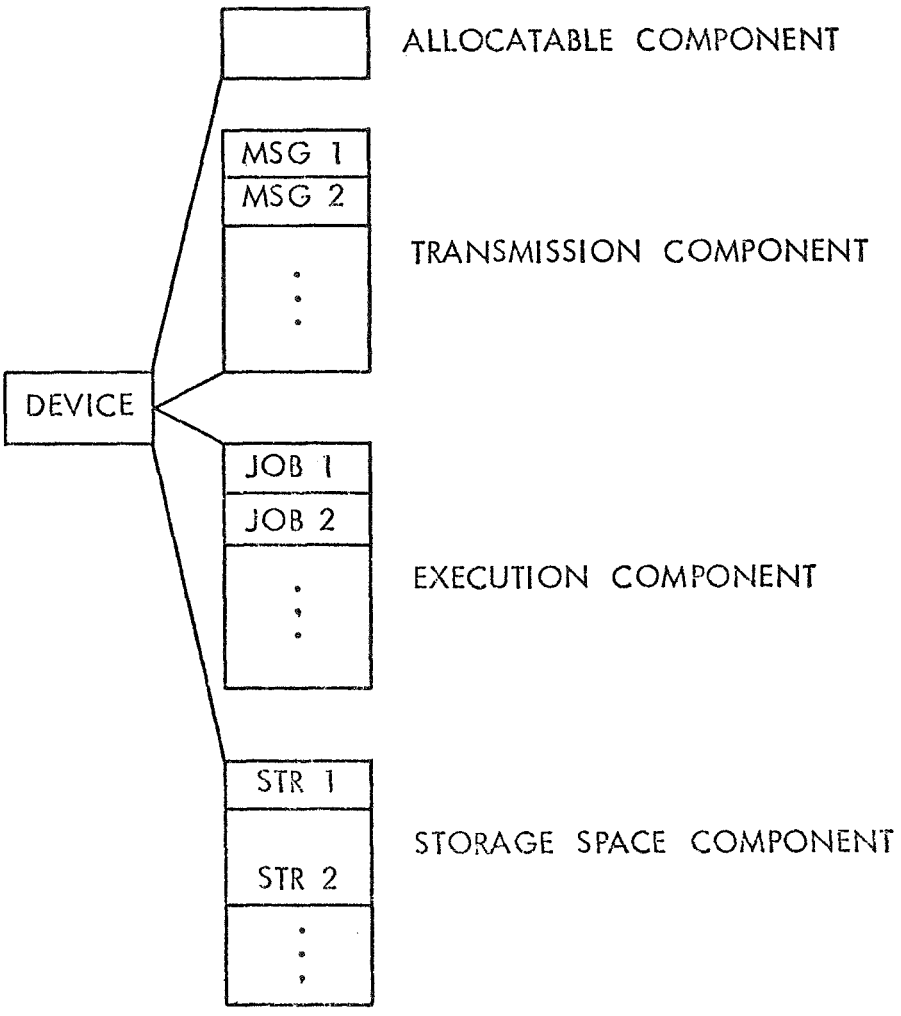

Fig. 1--ECSS Device Components

intended purpose. Some of these parameters are: 1) for Storage Space, the maximum amount; 2) for Execution, the instruction processing rate and maximum number of jobs that may be processed concurrently; 3) for Allocatable, whether a device may be allocated to more than one job simultaneously; and 4) for Transmission, the maximum data rate, number of simultaneous data streams that may be transmitted, delay per transmission, and others. There are also several parameters of interaction between components. Two of these

are software overhead times, used to model the software execution necessary to initiate data transmission or device allocation, and degradation factors, used to account for execution interference due to data transmission.

Besides characterizing individual devices, the System Description may define groups of devices. Groups are used for compact description (e.g., defining several devices having the same characteristics) and for automatic device selection at run time. Device selection is a feature of the built-in ECSS "operating system" that allows a group of devices to be put in a load command. This indicates that the actual device picked to handle the command should be the one available at any particular time during the run. The last function of the System Description is definition of certain data transmission paths within the system.

LOAD DESCRIPTION

The Load Description specifies the system's dynamic behavior. The load on a computer system is the work it must do, where "work" means the utilization of device components for certain time spans. Special programs in this section, called *jobs*, simulate the work of real application and control program processing by indicating sequences of hardware utilization commands.

Nearly any kind of deterministic or random effects may be included in a job to describe which activities are to be done, for how long, and in what order. The *sequence* is controlled by the testing, branching, looping, and other logical properties of the job. The simulated *time* at which a command is issued depends on the time necessary to do the work of the previous commands, and on any conditional delays within the job. Appendix A presents a general example of the progress of simulated time in a job.

The logical behavior of jobs is provided by appropriate SIMSCRIPT II statements. SIMSCRIPT and ECSS statements may be

freely intermixed in job descriptions. In fact, the full capability of this general simulation language (described in Kiviat, Villanueva, and Markowitz, 1968 [2]) is available as a subset of ECSS. New data structures may be created and used in the model, for example, to expand the description of the state-of-the-system or new routines written to incorporate complex system-control logic.

Simulation time in processing ECSS commands results from two situations. First, the use of certain device components is explicitly time-consuming; for example, executing instructions as directed by an EXECUTE command, and data transmission as specified by a SEND/RECEIVE statement. Second, some commands may or may not allow time to pass, depending on the state-of-the-system. Examples are conditional delays, indicated by WAIT or HOLD statements, which depend on whether the condition has come true at the time the statement is processed. Requests for storage space or for device allocation, GET and ALLOCATE respectively, may cause delay because that component is already being utilized, and the job must wait for it to become free. Execution or transmission commands may also involve this kind of delay if those components are busy. Priority-ranked queues are associated with each component of a device to keep track of pending requests. Marking the end of utilization of a component, e.g., with a FREE space or a DEALLOCATE statement, usually does not hold up job processing.

Several other load commands take zero time, but serve to define the jobs and their initiation. These include JOB--marking the beginning of a job, LAST--marking the end, START--commanding job initiation, and INITIALLY START--directing exogenous job-starting to simulate the system environment.

SERVICE ROUTINES

The Service Routines take care of the housekeeping details of internal model control. A collection of SIMSCRIPT II subprograms is used by ECSS to implement the actions specified by the Load Description. The event-scheduling and process-pointer management necessary to realize the flow-orientation of jobs within a SIMSCRIPT II context are included. All the system-state updating associated with the interaction of jobs and devices, jobs and jobs, and devices and devices are incorporated in these subprograms.

Moreover, this package of routines supplies a number of such operating-system functions as resource allocation by priority; I/O interrupt handling; device selection from groups; and queueing, dequeueing, and retrial of device requests. These capabilities allow easy specification of multiprogramming, multiprocessing, real-time processing, and conversational transmissions. The ECSS user automatically gets these capabilities when defining his model's dynamics.

Appendix B describes a complete ECSS simulation, which illustrates the use of ECSS in modeling a specific system, and shows the form and structure of a typical ECSS program. It also demonstrates the usage of SIMSCRIPT II statements within ECSS.

III. ADVANTAGES OF THE ECSS APPROACH

NATURAL INPUT LANGUAGE

Both ECSS and SIMSCRIPT II statements are English-like, and their procedural format allows a clear and flexible design. The user may incorporate his own definitions for various dimensional units. Considerable freedom for mnemonic names of devices, jobs, variables, events, etc., is allowed.

Use of a natural, computer-system-oriented input language also enhances ECSS as a communication and documentation tool. Explaining an ECSS model is much easier than explaining one written in an assembly-style language, or a higher-level language not specifically designed for computer system models.

PROVISION OF DECLARATIONS AND COMMANDS FOR COMMON COMPUTER SYSTEM OPERATIONS

Hardware elements are compactly defined and described in the System Description, and a variety of utilization commands for describing loads are provided. Requests for storage space, job starting, execution, and data transmission require only single statements. The Service Routines assume much of the modeling burden by handling the details and providing the built-in operating-system capabilities.

FLEXIBILITY

ECSS provides a variety of techniques for modeling systems. Both flow-oriented jobs and events can operate on the system state, depending on the modeler's preference. Jobs can model program behavior, system input characteristics, or arbitrary activities running on any appropriate device. The generality of devices, each having as many as four components, allows the modeling of nearly any kind of equipment. No particular structures are forced on the user.

No particular level of detail is required of the model. The example in Appendix B focuses on job-scheduling software including space reservation, space release, and necessary execution time, whereas I/O interrupt handling, transmission path selection, multiprogramming, and other control operations are left up to built-in ECSS functions. In other models, these operations may be of interest and could be modeled explicitly. Changes in detail are also readily incorporated into a given model.

EXTENDABILITY

ECSS and SIMSCRIPT statements are used in conjunction to build simulation models. Provision of the data types and procedural statements of SIMSCRIPT II allows the user to go beyond the primary ECSS capabilities for any purpose. In addition, the user may extend the definition of a device, a job, a transmission, or any other ECSS structure by appropriate SIMSCRIPT preamble statements that add attributes to these entities. New commands consisting of combinations of SIMSCRIPT and ECSS statements may be defined as another means of extending ECSS capability.

MODIFIABILITY

For some simulation models, the user may wish to change certain ECSS operations. A model may require different disciplines for certain queues, or different job suspension/reactivation criteria than are included in the standard ECSS package. Such changes would require modification of one or more of the Service Routines. This task is aided by both the modularity of the Service Routines and their description in SIMSCRIPT II. The source code for any of these routines is open to change. Of course, this requires a fairly good knowledge of the internal working of ECSS, but the clarity of the SIMSCRIPT code makes change much easier than if the user had to cope with assembly language.

RERUN ECONOMY

Simulation models are usually rerun a number of times, both during development (to debug, test, and validate the model) and in production (to investigate behavior under various conditions). ECSS is constructed to avoid recompilation of the model for each run. First, it produces a summary deck of the static system description, which allows changing system parameters without reprocessing the System Description. Second, the object decks representing the jobs, events, functions, and other routines written by the user are available for subsequent runs. Finally, the system and load are independent of each other with respect to system parameters (e.g., CPU speed, number of disks, etc.). This allows different systems and loads, which were not originally processed together, to be combined later as summary decks and object modules, again avoiding the overhead of recompilation.

IV. WEAKNESSES OF ECSS

ECSS is weak in some areas in the sense that certain capabilities are less convenient or less flexible than they should be. Although it is possible to model nearly any system in ECSS with the help of SIMSCRIPT II, the user must still work harder than he should have to for certain classes of operations. Implementation aspects, e.g., running-speed and core-storage requirements, are of secondary importance at this stage, and are not discussed here.

The difficulty in coping with weaknesses varies considerably. Least difficult is writing extra SIMSCRIPT II routines, not involving ECSS system variables, to include some operation not specifically in the ECSS language. Manipulating ECSS system variables outside the Service Routines is moderately difficult because a knowledge of the function and use of these variables within those routines is required. Most difficult is changing the Service Routines because this demands an intimate knowledge of their working and interaction. We describe examples of some noticeable problems.

NEGLECTED STATEMENTS AND CAPABILITIES

Certain features should be included in a computer system simulator that have not been incorporated in this first version of ECSS. Formulating a complete set of these features is still difficult, but a few have become apparent. For example, although all data are available to the user, no statistics are automatically collected or reported by ECSS. Such things as device utilization, average waiting time in queues, and average queue length are nearly always of interest in computer simulations. These statistics should be collected and perhaps produced on demand as a summary report. Also convenient would be a summary of the static system simulated because the System Description may

be quite complex if hierarchical groups are used. Another desirable feature would be a built-in technique for stopping and restarting a model at any point, perhaps to allow parametric changes, or to account for initialization periods. There is, furthermore, very little consistency checking of the model at translation time or run time. ECSS should do more to flag self-contradictory or illogical System Descriptions.

Several common computer operations now must be modeled in SIMSCRIPT II, and probably deserve specific ECSS statements to handle them. A statement to change job priorities during their run would contribute convenience and clarity to a model, particularly when simulating such operating-system functions as task scheduling or interrupt masking. Another missing capability is data-file placement on storage devices for file-access modeling. Although files may now be placed on specific devices, or allocated from groups of devices, it would be quite handy to have statements indicating file position on a device for calculation of variable access times for sequences of transmissions from different files on the same device. A third deficiency is lack of built-in polling operations. All service requests are handled in a priority-interrupt fashion in ECSS. However, many applications require these requests to be handled differently. For example, remote terminals are often polled in a particular order to determine their transmission status, rather than each signaling the receiver through an interrupt mechanism. Incorporation of arbitrary (or changing) servicing order in models would be expedited by ECSS statements that specifically simulate polling.

HARD-TO-GET-AT MECHANISMS

Control of mechanisms invoked only by declaration is sometimes desired in the Load Description. One instance is

the degradation of instruction execution rate as declared by a DEGRADES clause in the System Description. Reductions in execution rate may occur for other reasons than transmission interference. Multiple CPUs contending for the same core memory will generally not run as fast as if only one CPU had access to the memory. The degradation mechanism could easily handle such cases if the user had more direct control over it when specifying jobs.

Software overhead is another declarative feature that could be more accessible. In the batch model, allocation takes 50 msec of overhead time, but a variety of other activities--job initiation, task switching, or core storage reservation--may also take time that should be counted as overhead. Hence, the user should be able to indicate overhead time for a variety of operating system functions, or perhaps at any time he wishes, to realize greater utility of the overhead accumulation procedures.

OVER-AUTOMATICITY

The user may also need greater control over the functions of Load Description statements. Sometimes these statements automatically do more than the user desires. When acquiring simulated storage space, for example, the ECSS GET statement not only finds and reserves space, but also causes the requesting job to wait for space if enough is not available at request time. One may instead want to return to processing with a note indicating an unsuccessful request if sufficient space is not available. That is, one may wish only to use those parts of GET that find the maximum amount of space available and compare it to the amount required, and not the part that queues unsuccessful requests for retrial.

A similar problem also occurs when SENDing data. It is possible to specify that a message be conveyed from one

job to another by means of a data transmission. These messages are often used to activate further processing of jobs waiting for them. Instead of terminating and re-starting a job, for example, it may be more realistic to suspend and reactivate it by means of a "clock interrupt." This involves only the signaling features of the SEND statement, all communication being within a processor, but there is no way of simply signaling without all the path selection and other data transmission machinery. Other cases could be mentioned that illustrate a need to use only part of the power of a Load Description statement.

AWKWARD CONTROL-PROGRAM REPRESENTATION

Although the automatic operating system is a great help in some models, the distribution of control functions between the Service Routines and user-written jobs makes for a sometimes strained relationship between the model and the real system being simulated. Jobs simulating control-program functions may not interface smoothly with the default ECSS system, particularly if different algorithms are desired for other kinds of queue handling, resource management, or for controlling the order of occurrence of internal model events. Low-level changes in these operations usually require extensive changes both to the Service Routines and to the jobs representing the load's simulated software. Moreover, the code implementing the new operating-system functions may be scattered over parts of several routines, thereby decreasing model clarity. It would be desirable to keep most or all of user-specified control-program models in one place, either through a number of "operating-system-jobs" or some other unifying concept.

V. DESIGN CONSIDERATIONS FOR FURTHER ECSS DEVELOPMENT

Pointing out weaknesses in something under development is usually tantamount to saying how it will be improved. Reflection on these weaknesses has indicated some general principles in designing computer system simulation languages that may be used to anticipate and correct difficulties before they intrude into some new application of the language.

DECLARATIVE VERSUS PROCEDURAL DESCRIPTION

Although declarative specification of system features is most convenient, procedural specification is more flexible. Because one cannot anticipate all possible types of interaction, all interaction mechanisms should be accessible to procedural control so that the user can make the fullest use of provided capabilities. Declarative specifications should be retained, however, for their convenience, and perhaps even expanded to further parameterize the built-in operating system (e.g., more opportunities for overhead time specification).

STATEMENT SCOPE

Just as one cannot anticipate all possible interaction types, one cannot foresee all the combinations of ways to command the system components. Variable sophistication of detail requires greater user control over the actions of the Load Description statements. Powerful statements, incorporating a number of operations in a predetermined sequence, are necessary to quickly develop coarse models. For more detailed models, each operation should be available separately. Control at a lower functional level allows the same Service Routine mechanisms to be arranged in a greater number of ways without the chore of altering

their structure. The inclusion of more optional clauses for statements may ease the transition from coarse to fine levels of detail during model development.

OPERATING-SYSTEM ACCESSIBILITY

Service Routine alteration could also be avoided by inclusion of a number of exit points to user routines. Such a "monitor" is now included, but operates as an observer telling what the system did, not what it is about to do. Substituting an active monitor, with the power to skip some of the built-in operations on command, would provide more flexible use of the ECSS operating-system features.

AUTOMATIC SUMMARIES

A computer system should include automatic collection of statistics, and automatic output of statistical and other summaries, in addition to user access to all operations data. Certain quantities (e.g., device utilization, queue lengths, etc.) are of known importance, and statistics on them should be collected. Preformatted reports of these statistics would provide a convenient overview of model operation. Machine- and man-readable system structure summaries could further make the simulation clearer to the user, and aid in rerunning the model. Trace output should also be available on demand (perhaps through the monitor), but undesired volumes of operations data should be avoided.

VI. CONCLUSION

Use of the initial version of ECSS has shown it to be a convenient and powerful analysis tool. Its provisions for describing both common computing system elements and operations ease much of the modeling burden; its extendability and modifiability insure suitability for uncommon applications. Further developments on the prototype are proceeding as outlined in this Report.

Appendix A

ECSS JOB PROCESSING

An ECSS job is a specification of a sequence of activities. An activity is the utilization of a simulated device for some amount of simulated time as directed by a hardware utilization command. Hence, the simulation clock may advance during the processing of a job. Figure 2 illustrates the concept of job processing and the effect of logical statements and conditional delays on the progress of simulated time within a job.

One may think of a process-pointer moving through a job, indicating which statement is being processed. In Fig. 2, the job SHOW.TIMING is initiated at t_o, and the pointer starts at the first statement. This happens to be a command that takes time t_1, so processing does not proceed to the next statement until time $t_o + t_1$. Logical statements may then redirect the process-pointer: If VARIABLE > 0, then processing jumps to the point labeled 'NEXT' in the job (right-hand time column); otherwise, Commands 2 and 3 are processed (left-hand time column), requiring $t_2 + t_3$ more simulated time to pass. Conditional delays may suspend job processing. The process-pointer will not go beyond the HOLD statement until "input" has arrived at t_{in}, but if input has already arrived, processing will proceed directly to Command 4. This flow-orientation of jobs, resembling GPSS [3], is extremely useful in modeling computer system loads.

Simulated time at beginning of command processing		Job description	Comments
. t_0		JOB TO SHOW . TIMING	
t_0		Command 1	(takes time t_1)
$t_0 + t_1$		IF VARIABLE > 0, GO TO NEXT ELSE	
VARIABLE $\leq$ 0	VARIABLE > 0		
$t_0 + t_1$	—	Command 2	(takes time t_2)
$\sum_{i=0}^{2} t_i$	—	Command 3	(takes time t_3)
$\sum_{i=0}^{3} t_i$	$t_0 + t$	'NEXT'	(takes no time)
$\sum_{i=0}^{3} t_i$	$t_0 + t_i$	HOLD UNTIL INPUT ARRIVES	(conditional delay — assume input arrives at $t_{in} > t_0 + t_1$)
t_m	t_{in}	Command 4	(takes time t_4)
$t_m + t_4$	$t_{in} + t_4$	Command 5	(takes time t_5)
$t_m + t_4 + t_5$	$t_{in} + t_4 + t_5$	LAST	
where $t_m = \max\left(\sum_{i=0}^{3} t_i, t_{in}\right)$			

Fig. 2--ESS Job Processing

Appendix B

AN EXAMPLE OF THE ECSS APPROACH

This appendix presents a simplified model of a multiprogrammed batch-processing system to illustrate the structure and operation of an ECSS simulation program. This example has not been constructed to demonstrate good system design.

Figure 3 shows the system hardware configuration. A processor connects through three I/O ports to a cardreader and two controllers that in turn are connected to four disks. The system's load consists of disk file updating routines, a sequence of which are entered by means of the cardreader. The flow of tasks through the system is diagrammed in Fig. 4. In addition, some simulated software is included in the load to schedule jobs according to their space requirements and to handle disk allocation.

Figure 5 lists the model's entire simulation program. Some comments appear in parenthesis; each statement has been numbered; and SIMSCRIPT statements are marked with an (S). Besides the System and Load Descriptions mentioned above, the listing shows a preamble, a definition-description section, some events, and the MAIN routine. The preamble (statements 1-11) defines global variables and other data structures to be used in the model. The definition section (12-15) is an additional ECSS element that allows relation of user-defined terms to basic ECSS terms (with possible conversion factors) for use in the System and Load Descriptions. Because no statistics are collected or reported by ECSS, the event OBSERVATION (24-29) is used in this program to view the system every five seconds, and to print a record of its activity. QUITSIM (statement 30) halts the simulation. Finally, the MAIN routine (31-35) indicates when to stop and when to make the first observation. It then directs the simulator to commence.

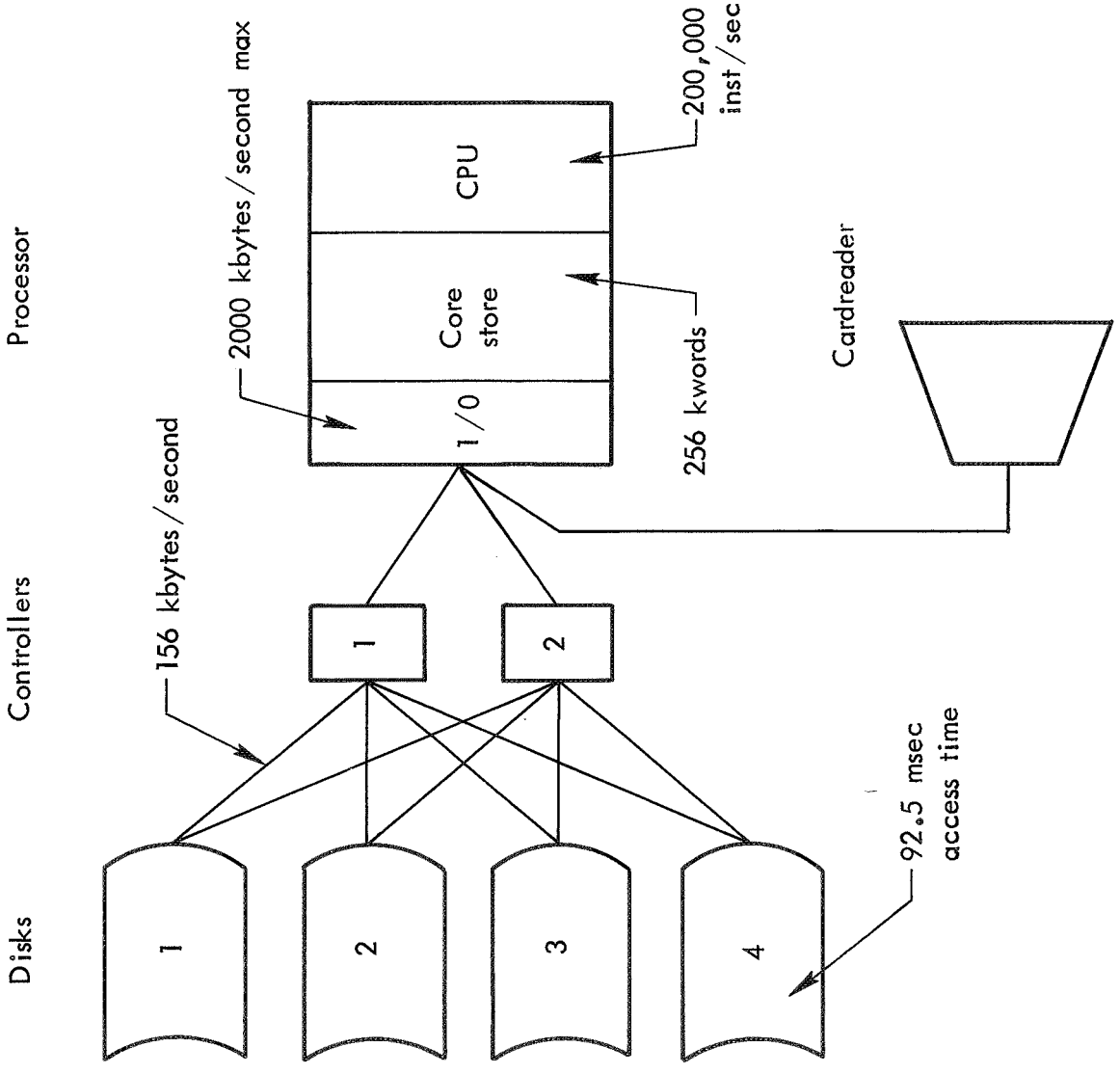

Fig. 3--Model Hardware Configuration

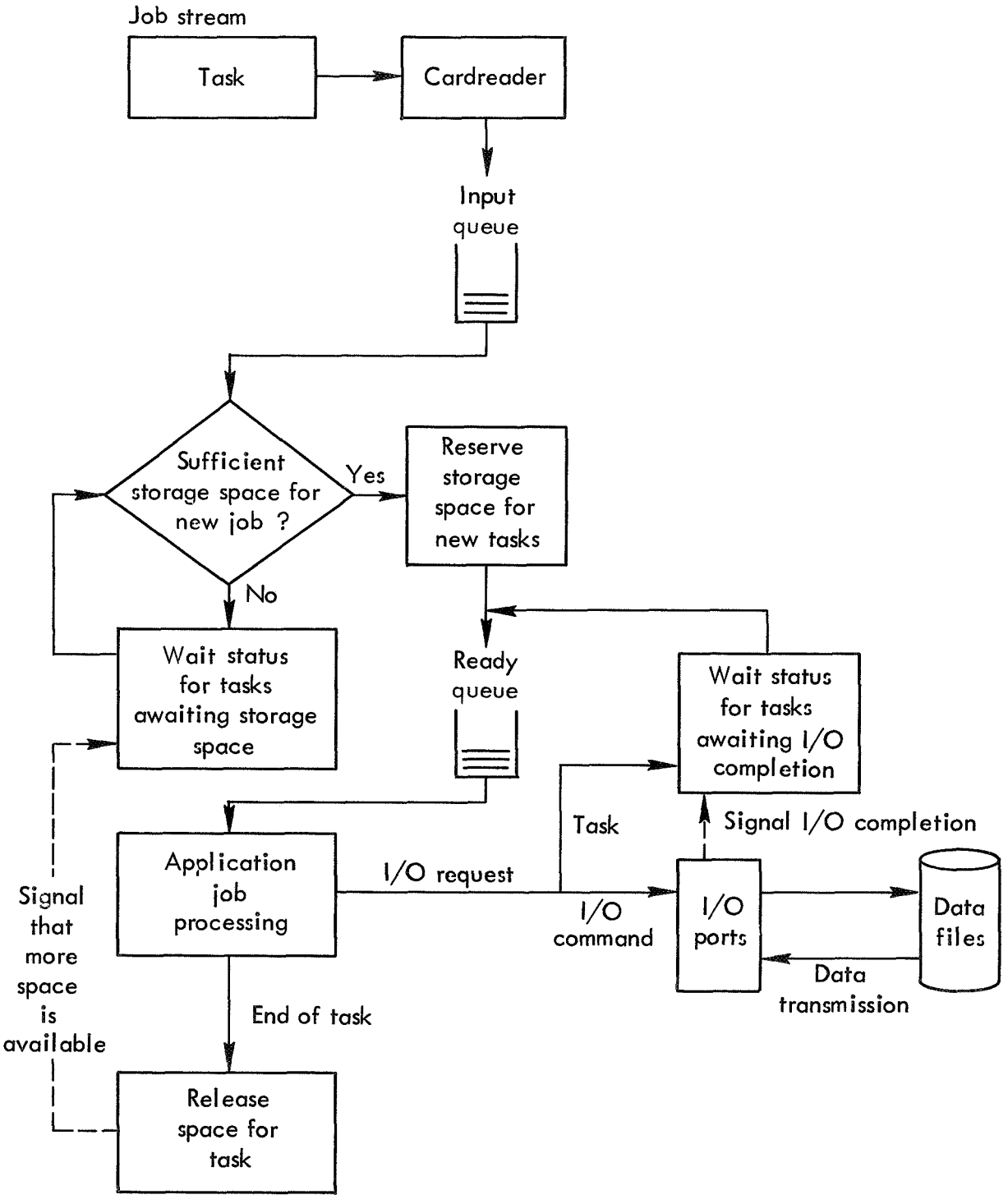

Fig. 4--Work Flow Through Multiprogrammed Batch Processor

```
 1.  (S) PREAMBLE (defines global variables and data structures)
 2.  (S)   DEFINE TOTAL.IN.TIME AS A REAL VARIABLE
 3.  (S)   DEFINE JOBS.STARTED, JOBS.COMPLETED, JOBS.IN.PROCESS
             AS INTEGER VARIABLES
 4.  (S) TEMPORARY ENTITIES
 5.  (S)   EVERY JOB.DESCRIP HAS A SPACE.REQMT, A LOOP.REQMT
             AND A TIME.IN AND BELONGS TO THE JOB.QUEUE
 6.  (S)   DEFINE SPACE.REQMT, TIME.IN AS REAL VARIABLES
 7.  (S)   DEFINE LOOP.REQMT AS AN INTEGER VARIABLE
 8.  (S)   THE SYSTEM OWNS A JOB.QUEUE
 9.  (S)   DEFINE JOB.QUEUE AS A SET RANKED BY HIGH SPACE.REQMT
10.  (S) EVENT NOTICES INCLUDE OBSERVATION,QUITSIM
11.  (S) END

12.      DEFINITION DESCRIPTION (incorporates user terminology)
13.        DEFINE UNITS KBYTES=1000 TRANSMISSION.UNITS,
             KWORDS=SPACE.UNIT
14.        DEFINE UNITS SUBROUTINES=400 INSTRUCTIONS
15.      END

16.      SYSTEM DESCRIPTION (defines characteristics of each
           class of devices)
17.        SPECIFY 1 PROCESSOR,
             EXECUTES 200 INSTRUCTIONS PER MILLISECOND
               (execution rate)
             TRANSMITS 2000 KBYTES PER SECOND (transmission rate)
             HAS CAPACITY OF 3 TRANSMISSION USERS (models
               3 subchannels)
             CONNECTS TO CONTROLLERS
             DEGRADES PROCESSOR BY 10% PER 200 KBYTES/SECOND
               (models cycle-stealing)
             HAS CAPACITY OF 256 KWORDS (storage space)
             ALLOCATES DISKS IN 50 MS (models gross software effect)

18.          SPECIFY 2 CONTROLLERS, EACH
               HAS CAPACITY OF 1 TRANSMISSION USER
               CONNECTS TO DISKS,PROCESSOR

19.          SPECIFY 4 PUBLIC DISKS, EACH
               HAS CAPACITY OF 1 TRANSMISSION USER
               ABSORBS 92.5 MILLISECONDS PER MESSAGE (access time)
               TRANSMITS 156 KBYTES PER SECOND (disk transmission rate)
               CONNECTS TO CONTROLLERS

20.          SPECIFY 1 CARD.READER, CONNECTS TO PROCESSOR

21.          PATH OUTPATH IS PROCESSOR,CONTROLLERS, DISKS
22.          PATH INPATH IS DISKS,CONTROLLERS,PROCESSOR
23.      END

24.  (S) EVENT OBSERVATION SAVING THE EVENT NOTICE
25.  (S)   RESCHEDULE THIS OBSERVATION IN 5 UNITS (seconds)
26.  (S)   LET AVG.TURNAROUND=TOTAL.IN.TIME/JOBS.COMPLETED
             (calculate statistics)
27.  (S)   LET AVG.THRUPUT=JOBS.COMPLETED/TIME.V
28.  (S)   LIST TIME.V,JOBS.STARTED,JOBS.IN.PROCESS,JOBS.COMPLETED,
             AVG.TURNAROUND,AVG.THRUPUT (print out statistics)
29.  (S) END

30.  (S) EVENT QUITSIM  STOP END
```

Fig. 5--Batch Processor Simulation

```
31.  (S) MAIN
32.  (S)   SCHEDULE A QUITSIM AT 35
33.  (S)   SCHEDULE AN OBSERVATION AT 5
34.  (S)   START SIMULATION
35.  (S) END

36.      LOAD DESCRIPTION
           (defines sequences of system utilization commands)
37.      JOB READER
38.  (S)   CREATE A JOB.DESCRIP
39.  (S)   LET SPACE.REQMT(JOB.DESCRIP)=EXPONENTIAL.F(104.0,1)
40.  (S)   READ LOOP.REQMT(JOB.DESCRIP)
41.  (S)   LET TIME.IN(JOB.DESCRIP)=TIME.V
42.  (S)   ADD 1 TO JOBS.STARTED
43.  (S)   FILE JOB.DESCRIP IN JOB.QUEUE
44.        START JOB READER ON CARD.READER
             IN EXPONENTIAL.F(12.0,1) SECONDS
45.        LAST

46.      JOB INITIATOR
47.  (S)   IF JOB.QUEUE IS EMPTY,
             START JOB INITIATOR ON PROCESSOR WITH PRIORITY 2 IN 1 SECOND
49.  (S)     RETURN
50.  (S)     ELSE
51.  (S)   REMOVE THE FIRST JOB.DESCRIP FROM JOB.QUEUE
52.  (S)   ADD 1 TO JOBS.IN.PROCESS
53.        EXECUTE 100 INSTRUCTIONS (space reservation processing)
54.        GET SPACE.REQMT (JOB.DESCRIP) CONTIGUOUS KWORDS FROM PROCESSOR
55.        START JOB INITIATOR ON PROCESSOR WITH PRIORITY 2
56.        EXECUTE 500 INSTRUCTIONS (job initiation processing)
57.        START JOB APPLICATION(LOOP.REQMT(JOB.DESCRIP)) ON PROCESSOR
             WITH PRIORITY 1 WAITING HERE FOR COMPLETION
58.        FREE SPACE.REQMT(JOB.DESCRIP) KWORDS FROM PROCESSOR
59.  (S)   SUBTRACT 1 FROM JOBS.IN.PROCESS
60.  (S)   ADD 1 TO JOBS.COMPLETED
61.  (S)   ADD TIME.V-TIME.IN(JOB.DESCRIP) TO TOTAL.IN.TIME
62.  (S)   DESTROY THE JOB.DESCRIP
63.        LAST

64.      JOB APPLICATION (REQD.LOOPS)
65.        DEFINE REQD.LOOPS AND L AS INTEGER VARIABLES
66.        ALLOCATE DISKS# 1 AS INPUT.FILE
67.        ALLOCATE DISKS AS OUTPUT.FILE
68.  (S)   FOR L=1 TO REQD.LOOPS DO ...
69.          RECEIVE RECORD ''BLOCK OF DATA'' OF LENGTH 800 FROM
               INPUT.FILE VIA INPATH WAITING HERE FOR COMPLETION
               (read data)
70.          EXECUTE 10 SUBROUTINES (process data)
71.          SEND RECORD OF LENGTH 160 TO OUTPUT.FILE VIA OUTPATH
               WAITING HERE FOR COMPLETION (write data)
72.  (S)     LOOP
73.        DEALLOCATE INPUT.FILE
74.        DEALLOCATE OUTPUT.FILE
75.        LAST

76.      INITIALLY START READER ON CARD.READER (initialize the system)
77.      INITIALLY START INITIATOR ON PROCESSOR WITH PRIORITY 2
78.      END
```

Fig. 5--Continued

MODEL OPERATION

The ECSS job READER (37-45) models the input environment of the system. This consists of units of work submitted at exponentially distributed times, with a mean of 12 seconds. The work is characterized by its space requirement, selected from an exponential distribution with a mean of 104 (KWORDS) and a duration parameter, represented as the number of simulated processing cycles, which is read in from a data card. Work to be done is placed in a queue, JOB.QUEUE, and ranked on its space requirement from high to low. SIMSCRIPT II provides the definition and manipulation statements for this queue. The use of JOB.QUEUE augments the default ECSS job-scheduling procedures.

The prototype for the work to be done is the APPLICATION job (64-75), which runs on the processor. First, one disk is designated as the input source and one of the disks is selected randomly (by the built-in ECSS operating system) as the output unit by the ALLOCATE statements. This task is considered to require 50 msec of overhead time per allocation (from 17), and hence the processor's execution component is busy for 100 msec as well as the allocatable component of the disks being set. Because the disks are PUBLIC, more than one job can use them simultaneously.

Next, the APPLICATION job begins the simulated reading of blocks of data, processing of the data, and writing of an output record (69-71). The RECEIVE statement picks a free data path from the INPUT.FILE disk, through one of the controllers, to one of the I/O ports of the processor. The job automatically waits for a free path if none is available. Transmission components of these devices are then busy for the duration of the disk access time (92.5 ms), plus the time for transmission of 800 bytes at 156 kbytes/sec (disk speed). APPLICATION job processing is suspended for that time, as directed by the

WAITING clause (69). The processor execution component is then busy simulating the execution time for 10 sets of 400 instructions (SUBROUTINES) at 200,000 instructions/second. Finally, the SEND statement goes through a similar procedure as the RECEIVE to model writing the output record. The job's last action is to reset the allocation status of the disks it used, i.e., free them, which takes no simulated time in this model.

A scheduling algorithm is modeled in the INITIATOR job (46-63). It starts APPLICATION jobs in the order determined by the JOB.QUEUE, i.e., large jobs first (if the queue is empty, an INITIATOR tries again in one second). Execution time for scheduling is modeled with the first EXECUTE statement (53). The GET statement (54) reserves a contiguous block of the processor's storage space necessary to satisfy the space requirement. If not enough space is available, the INITIATOR job is suspended until space is available. When space is reserved, another INITIATOR is started (for the next JOB.QUEUE work unit) and some execution time, representing job initiation bookkeeping, is called for (55-56). Following that, an APPLICATION job is started on the processor, passing its duration as an argument. Upon completion, the INITIATOR proceeds to release the space for that unit of work (58), and terminates.

Several INITIATOR and APPLICATION jobs may be running concurrently on the processor in a multiprogrammed fashion. Contention for devices is resolved by priority (note that INITIATOR jobs with priority 2 will always get a device before an APPLICATION job), then by time of request. This is performed by the Service Routines. Scattered throughout the jobs are various SIMSCRIPT II statements that collect the data periodically reported by the OBSERVATION event. Figure 6 shows some typical output from this simulation program.

TIME.V	JOBS.STARTED	JOBS.IN.PROCES	JOBS.COMPLETED	AVG.TURNAROUND	AVG.THRUPUT
5.000000	2	1	1	2.116899	.200000
TIME.V	JOBS.STARTED	JOBS.IN.PROCES	JOBS.COMPLETED	AVG.TURNAROUND	AVG.THRUPUT
10.000000	2	0	2	2.250345	.200000
TIME.V	JOBS.STARTED	JOBS.IN.PROCES	JOBS.COMPLETED	AVG.TURNAROUND	AVG.THRUPUT
15.000000	2	0	2	2.250345	.133333
TIME.V	JOBS.STARTED	JOBS.IN.PROCES	JOBS.COMPLETED	AVG.TURNAROUND	AVG.THRUPUT
20.000000	2	0	2	2.250345	.100000
TIME.V	JOBS.STARTED	JOBS.IN.PROCES	JOBS.COMPLETED	AVG.TURNAROUND	AVG.THRUPUT
25.000000	2	0	2	2.250345	.080000
TIME.V	JOBS.STARTED	JOBS.IN.PROCES	JOBS.COMPLETED	AVG.TURNAROUND	AVG.THRUPUT
30.000000	2	0	2	2.250345	.066667
TIME.V	JOBS.STARTED	JOBS.IN.PROCES	JOBS.COMPLETED	AVG.TURNAROUND	AVG.THRUPUT
35.000000	4	1	2	2.250345	.057143

Fig. 6--Batch Model Program Output

REFERENCES

1. Neilsen, N. R., *ECSS: An Extendable Computer System Simulator*, The RAND Corporation, RM-6132-NASA, February 1970.

2. Kiviat, P. J., R. Villanueva, and H. M. Markowitz, *The SIMSCRIPT II Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1968.

3. *General Purpose Simulation System/360, User's Manual*, International Business Machines, Inc., Form H20-0326-2, 1968.